What is Spring?

- Light-weight yet comprehensive framework for building Java SE and Java EE applications.
- Initially built to reduce the complexities of Enterprise Java development.
- Based on POJOs and Interfaces
- Applies Inversion-of-Control principles, specifically using the Dependency Injection technique, which aims to reduce dependencies of components on specific implementations of other components.
- Integration with persistence frameworks like Hibernate.
- Extensive aspect-oriented programming (AOP) framework to provide services such as transaction management
- MVC web application framework, built on core Spring functionality, supporting many technologies for generating views, including JSP, FreeMarker, Velocity and Tiles.

History of Spring

- First conceived and developed by Rod Johnson
- Rod Johnson described Spring in his book One-on-One : J2EE Design and Development.
- Release 1.0 was in 2004
- Release 2.0 was released in 2006 supported AspectJ
- Annotations were introduced in 2.5, which was released in 2007
- 3.0 was released in 2009 supported Java Configuration
- 4.0 in Dec, 2013 supports Java SE 8 and Java EE 7
- Spring Boot 1.0 was released on 1-Apr-2014
- 5.0 on 28-SEP-2017 with support for Java 9
- Spring Boot 2.0 was released in Nov, 2017
- Spring Boot 2.7.6 was released in Nov, 2022.

Key Strategies

- Lightweight and minimally invasive development with POJOs
- Loose coupling through DI and interface orientation
- Declarative Programming through Aspects and common conventions
- Eliminating boilerplate code with aspects and templates

What is Spring Boot?

- Spring Boot is the starting point for building all Spring-based applications.
- Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring.
- Embed Tomcat, Jetty or Undertow directly (no need to deploy WAR files)
- Provide opinionated 'starter' dependencies to simplify your build configuration
- Automatically configure Spring and 3rd party libraries whenever possible
- Absolutely no code generation and no requirement for XML configuration

How to get started?

- Go to Spring Initializr (https://start.spring.io/), fill project details and pick options like Web, JPA etc. and download maven project with POM file in the form of .zip.
- Extract zip file into a folder.
- Import project from extracted folder into Eclipse as Maven Project.

POM.XML

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.0.0</version>
</parent>
<dependencies>
 <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
 </dependency>
</dependencies>
<properties>
    <java.version>1.17</java.version>
</properties>
<build>
  <plugins><plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId
        >
    </plugin>
  </plugins>
</build>
```

@SpringBootApplication

- Is an annotation that adds the following annotations :
  - ✔ @Configuration
  - ✔ @EnableAutoConfiguration
  - ✔ @ComponentScan

- This annotation is used for class that contains main()

SpringApplication Class

- Can be used to bootstrap and launch a Spring application from a Java main() method.
- By default class will perform the following steps to bootstrap your application:
  - ✔ Create an appropriate ApplicationContext instance (depending on your classpath)
  - ✔ Register a CommandLinePropertySource to expose command line arguments as Spring properties
  - ✔ Refresh the application context, loading all singleton beans
  - ✔ Trigger any CommandLineRunner beans
- In most circumstances the static run(Class, String[]) method can be called directly from your main method to bootstrap your application.

SpringApplication Methods

SpringApplication(Class<?>...primarySources) Create a new SpringApplication instance.

void addPrimarySources(Collection<Class<?>> additionalPrimarySources) Add additional items to the primary sources that will be added to an ApplicationContext when run(String...) is called.

void addPrimarySources (Collection<Class<?>> additionalPrimarySources)
Add additional items to the primary sources that will be added to an ApplicationContext when run(String...) is called.

void setBannerMode(Banner.Mode bannerMode)
Sets the mode used to display the banner when the application runs.

void setSources(Set<String> sources)
Set additional sources that will be used to create an ApplicationContext.

Application

```
@SpringBootApplication
public class MyApplication{

   // ... Bean definitions

   public static void main(String[] args) throws Exception {
        SpringApplication.run(MyApplication.class, args);
   }
 }
```

More Customized Application

```
@SpringBootApplication
public class MyApplication{

    public static void main(String[] args) throws Exception {
     SpringApplication application =
                new SpringApplication(MyApplication.class);
     // ... customize application settings here
     application.run(args)
  }
}
```

CommandLineRunner Interface

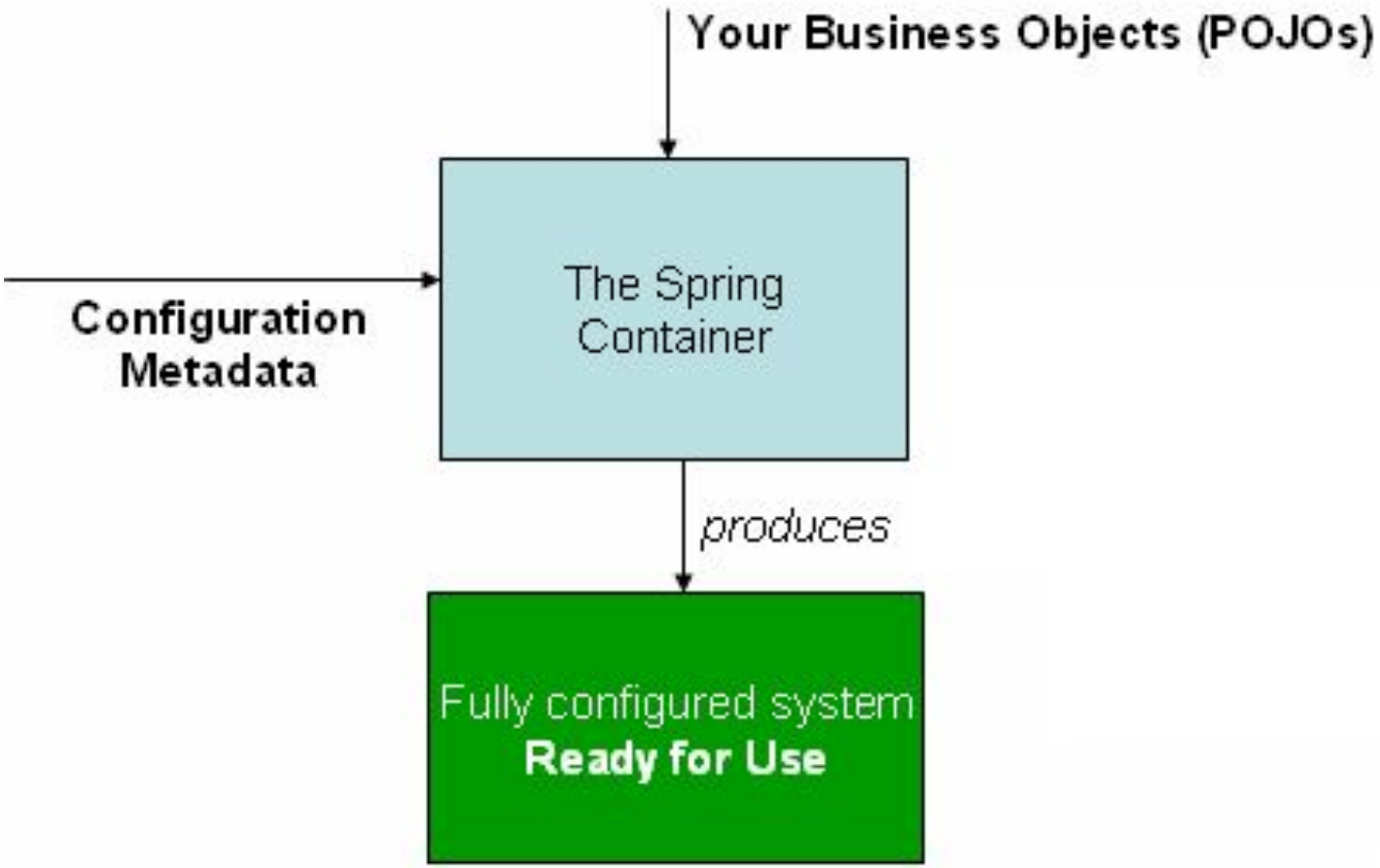
- Used to indicate that a bean should run when it is contained within a SpringApplication
- It contains a single method – run(String … args)
- SpringApplication.run() calls run(…) method of all beans that implement CommandLineRunner.

Features of Spring

- Dependency injection (DI) through IOC
- Aspect-Oriented Programming - Transaction management (AOP)
- Web applications and REST API
- Data access (JDBC, Hibernate)
- Security
- Messaging
- Testing
- More...

Spring IOC Container

- Provides Dependency Injection (DI)
- DI is the process whereby objects define their dependencies, that is, the other objects they work with, only through constructor arguments or properties that are set on the object instance after it is constructed
- The container then *injects* those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation


Your Business Objects (POJOs)
Configuration Metadata
The Spring Container
produces
Fully configured system
Ready for Use

Spring Beans

- The term “bean” is used to refer any component managed by the BeanFactory
- It is a POJO
- The “beans” are in the form of JavaBeans (in most cases) - no arg constructor and getter and setter methods for the properties
- Beans are singletons by default
- Properties may be simple values or references to other beans
- Beans can have multiple names
- These beans are created with the configuration metadata that you supply to the container

Bean Properties

- ❑ Name
- ❑ Scope
- ❑ Constructor arguments
- ❑ Properties
- ❑ Auto-wiring mode
- ❑ Lazy-initialization mode
- ❑ Initialization method
- ❑ Destruction method

Types of Dependency Injection

- Constructor based
- Property or field based
- Setter method based

```
// Field based DI
@Autowired
private Books books;
```

```
// Constructor based
DI  @Autowired
public Test(Books books)
    {  this.books = books;
}
```

```
// Setter based DI
@Autowired
public void setBooks(Books books) {
    this.books = books;
}
```

Auto Wiring using @Autowiring

- The Spring container is able to *autowire* relationships between collaborating beans.
- It is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the BeanFactory.
- Annotation @Autowired is used either with setter method or constructor to enable autowiring.
- If dependency is optional then use required property of Autowired annotation.
- By default auto wiring is done by type. So, if more than one bean is available for same type then it throws exception.

```
@Autowired(required=false)
private Catalog catalog;
```

@Qualifier

- This annotation may be used on a field or set method of property in candidate beans when autowiring.
- Used to qualify bean when two or more beans match auto wiring.
- Takes name of the bean as parameter.

```
@Component
public class Catalog {
    private  Books books;

    @Autowired
    @Qualifier("oracleBooks")
    public void setBooks(Books books) {
       this.books = books;
    }
    // code
}
```

```
@Component   // default name is oracleBooks
public class OracleBooks implements Books {}
```

Bean Scope

singleton (Default)

Scopes a single bean definition to a single object instance per Spring IoC container.

prototype

Scopes a single bean definition to any number of object instances.

request

Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition.

session

Scopes a single bean definition to the lifecycle of an HTTP Session.

@Scope annotation

- Specifies the scope of the bean
- Available options are prototype, singleton, request, session

```
@Component
@Scope(scopeName="prototype")
public class OracleBooks implements Books{

}
```

@PostConstruct and @PreDestory

- Specify initialization callback and destruction callback.
- A method carrying one of these annotations in a bean is invoked automatically by spring.
- In the example below, the cache will be pre-populated upon initialization and cleared upon destruction.

```
public class Catalog {
    @PostConstruct
    public void init()
    {
        // do the process needed after DI
    }
    @PreDestroy
    public void clearAll()
    {
       // clears resources before bean is destroyed
    }
}
```

Lazy-Initialized Beans

- By default, ApplicationContext creates and configure all singleton beans as part of the initialization process.
- When this behavior is *not* desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized.
- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
- Annotation @Lazy specifies the same for bean.

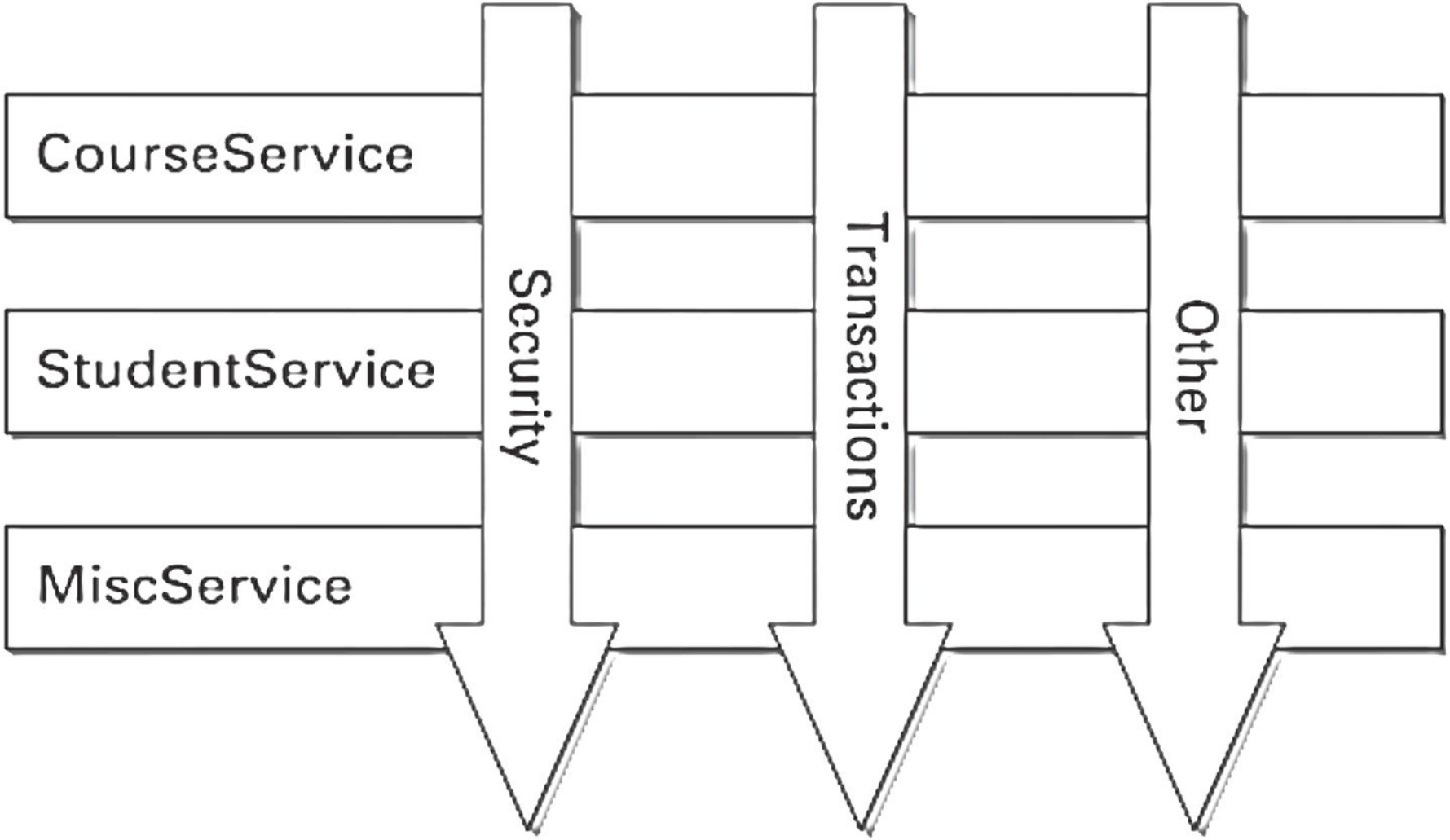
```
@Component
@Lazy
public class JavaBooks implements Books{

}
```

What is AOP?

- ***Aspect-Oriented Programming*** **(AOP) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure.**
- **The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*.**
- **Aspects enable the modularization of concerns such as transaction management that cut across multiple types and objects.**
- **AOP eliminates code tangling, where each method handles multiple concerns**
- **AOP also eliminates code scattering, where aspects are implemented in multiple methods.**

What is AOP?


CourseService
StudentService
MiscService
Security
Transactions
Other

Examples for Aspects

- Transaction Management
- Security
- Tracing
- Profiling
- Exception Management

Boot Starter for AOP

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

AOP Terminology

Aspect

A modularization of a concern that cuts across multiple classes. Transaction management is a good example of a crosscutting concern in enterprise Java applications. In Spring AOP, aspects are implemented using regular classes.

Join point

A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point always represents a method execution.

AOP Terminology

Advice

Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice.

Pointcut

A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. Spring uses the AspectJ pointcut expression language by default.

Target object

Object being advised by one or more aspects. Also referred to as the advised object. Since Spring AOP is implemented using runtime proxies, this object will always be a proxied object.

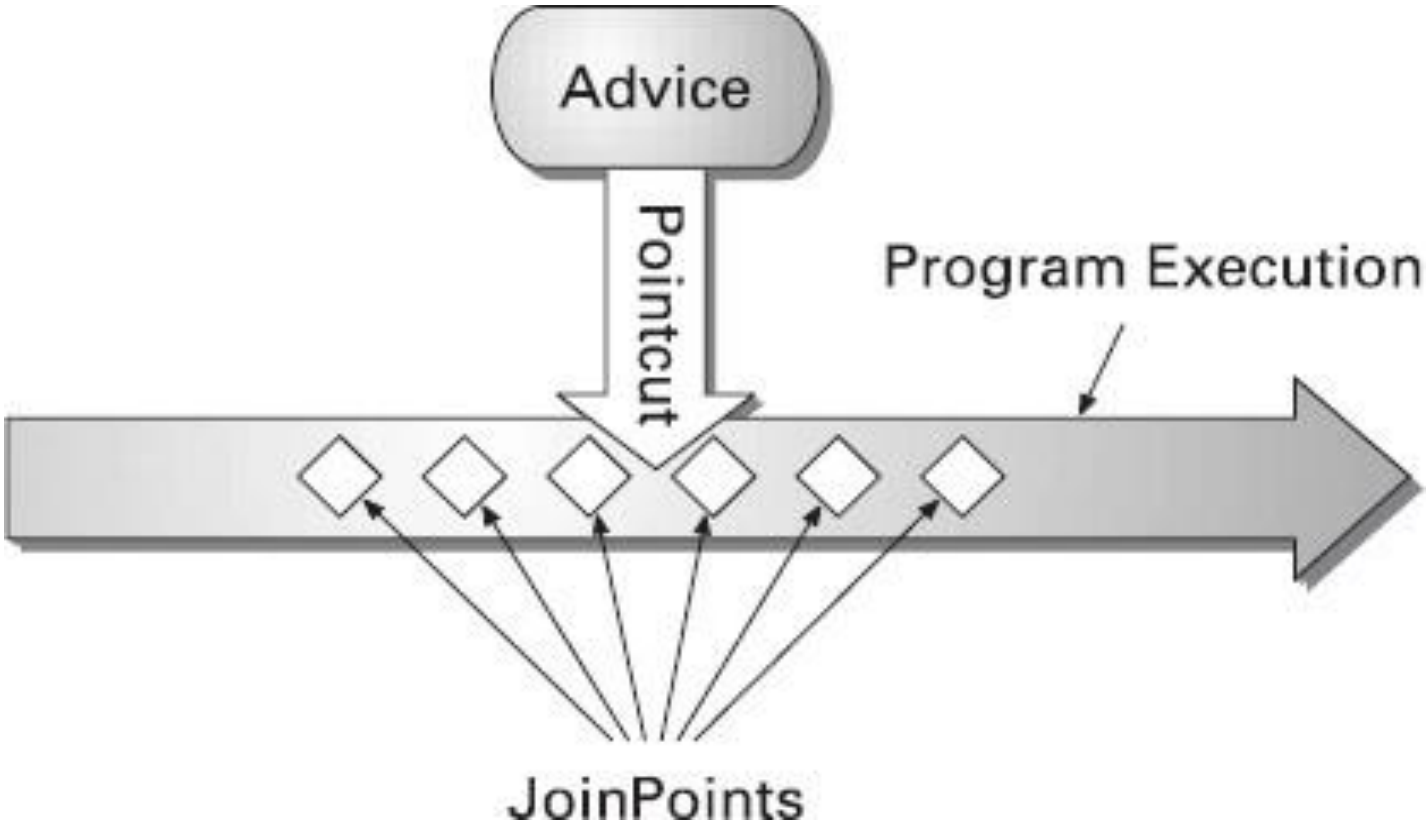
AOP Terminology

AOP proxy

An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.

Weaving

Linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.


Advice
Pointcut
Program Execution
JoinPoints

Types of Advices

Before advice

Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

After returning advice

Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

After throwing advice

Advice to be executed if a method exits by throwing an exception.

Types of Advices

After (finally) advice

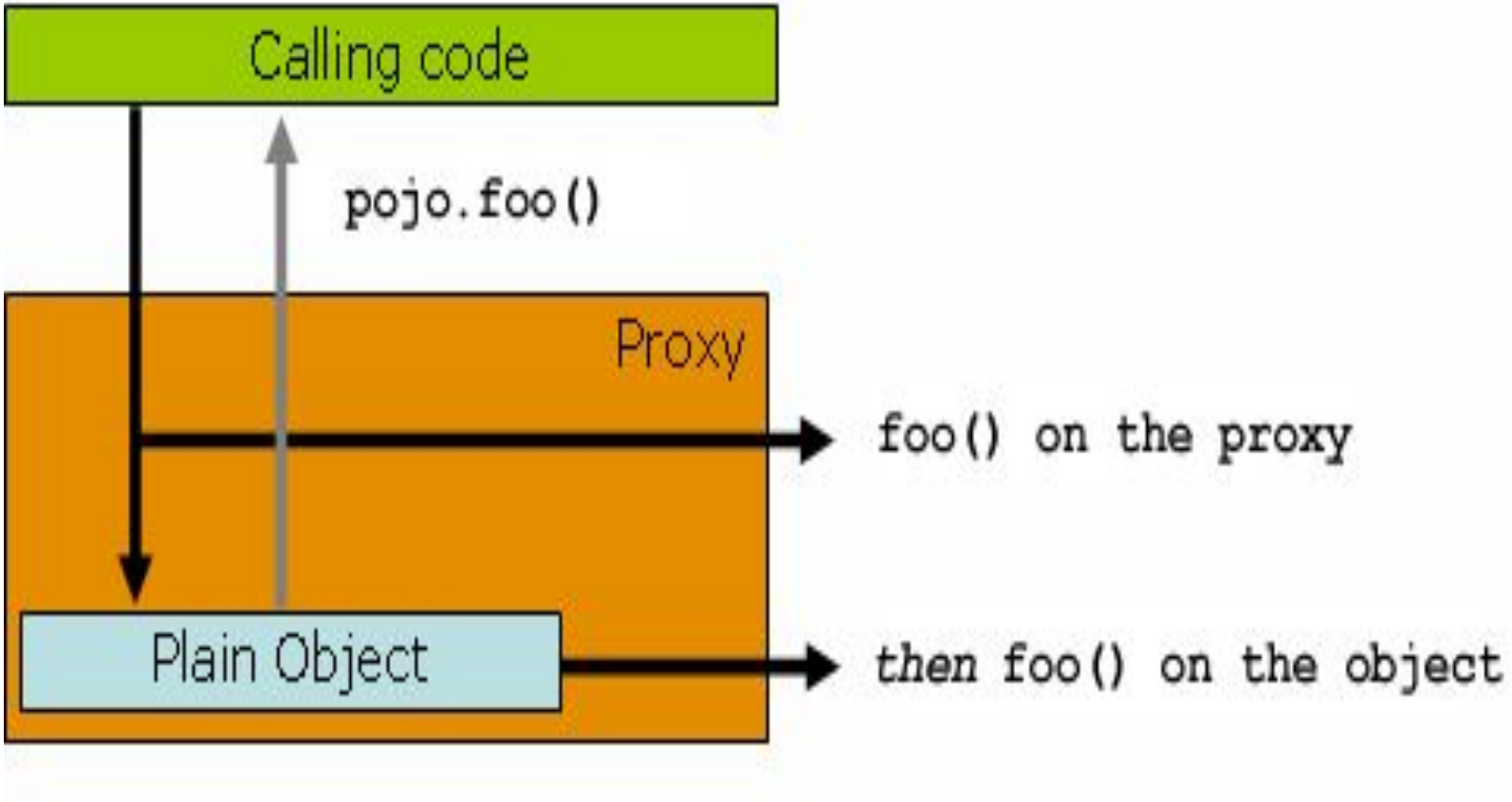
Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

Around advice

Advice that surrounds a join point such as a method invocation. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method.

AOP Proxies

- Spring AOP defaults to Java dynamic proxies for AOP proxies.
- If bean class implements at least one interface then it is uses JDK dynamic proxy.
- All of the interfaces implemented by the bean class will be proxied.
- If bean class doesn’t implement any interface then Spring AOP uses CGLIB proxies.


Calling code
pojo.foo()
Proxy
foo() on the proxy
Plain Object
then foo() on the object

@Pointcut

- Use @Pointcut annotation to define pointcuts.
- Method where Pointcut is declared must return void
- Pointcut has two parts – pointcut signature and pointcut expression
- Pointcut signature contains name and parameters
- Pointcut expression determines exactly which method executions we are interested in

Pointcut Syntax

```
Execution (modifiers-pattern? returning-type declaring-type?
          name-pattern (param-pattern) throws-pattern?)
```

- All parts except the returning type, name pattern, and parameters pattern are optional.
- Returning type pattern determines what the return type of the method must be in order for a join point to be matched.
- The name pattern matches the method name.
- The parameters pattern is slightly more complex
- Value () matches a method that takes no parameters, whereas (..) matches any number of parameters (zero or more).
- The pattern (*) matches a method taking one parameter of any type, (*,String) matches a method taking two parameters, the first can be of any type, the second must be a String.
- Operators and, or and not are also available.

Pointcut Designators(PCD) in Pointcut expression

execution

For matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP

within

Limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)

this

Limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type

Pointcut Designators(PCD) in Pointcut expression

target

Limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type

args

Limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types

Examples

The execution of any public method:
execution(public * *(..))

The execution of any method with a name beginning with "set" :
execution(* set*(..))

The execution of any method defined by the AccountService interface:
execution(* st.AccountService.*(..))

The execution of any method defined in the st package:
execution(* st..(..))

The execution of any method defined in the st package or a sub-package:
execution(* st...(..))

Pointcut Examples

Any join point within the st package:
`within(st.*)`

Any join point within the service package or a sub-package:
`within(com.xyz.service..*)`

Any join point where the proxy implements the Account interface:
`this(com.xyz.service.Account)`

Any join point where the target object implements Account interface:
`target(com.xyz.service.Account)`

Any join point which takes a single parameter, and where the argument passed at runtime is Serializable:
`args(java.io.Serializable)`

Pointcut Examples

```
@Pointcut("execution(public * (..))")
private void anyPublicOperation() {}

@Pointcut("within(com.st.trading..)")
private void inTrading() {}

@Pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}
```

JoinPoint

- A JoinPoint represents a method in your application where you can plug-in AOP aspect.
- A parameter of JoinPoint type can be used in an advice to get information about method that is being advised.

```
@Component
@Aspect
public class LogAspect {
    @Pointcut("execution(* catalog.Books.*(..))")
    public void allBooksMethods() {
    }

    @Before(value="allBooksMethods()")
    public void before(JoinPoint jp) {
       System.out.println("Before Advice for " +
                                jp.getSignature());
    }
}
```

JoinPoint Methods

Object[] getArgs()
Returns the arguments at this join point.
String getKind()
Returns a String representing the kind of join point.
Signature getSignature()
Returns the signature at the join point.
Object getTarget()
Returns the target object.
Object getThis()
Returns the currently executing object.
String toLongString()
Returns an extended string representation of the join point.
String toShortString()
Returns an abbreviated string representation of the join point.

Before and After advices

```
@Before(pointcut="...")
public void doBeforeProcess() {
}

// If we need to access method being advised
@Before(pointcut="...")
public void doBeforeProcess(JoinPoint jp) {
}
```

```
@After(pointcut="...")
public void doAfterProcess() {
}

// If we need to access method being advised
@after(pointcut="...")
public void doAfterProcess(JoinPoint jp) {
}
```

After returning and throwing advices

```
@AfterReturning(pointcut="...")
public void doAfterSuccess() {
}

// If we need to access value being returned by method
@AfterReturning(pointcut="...", returning="value")
public void doAfterSuccess(Object value) {
}
```

```
@AfterThrowing(pointcut="...")
public void doAfterFailure() {
}

// If we need to access value being returned by method
@AfterReturning(pointcut="...", throwing
="ex")  public void doAfterFailure(Exception
ex) {
}
```

Around Advice

```
@Around(pointcut="...")
public Object doProfiling(ProceedingJoinPoint pjp)
                throws Throwable {
     // before process
     Object retVal = pjp.proceed();
     // after process
}
```

Aspect Example

```
@Component
@Aspect
public class TraceAspect {
    @Pointcut("execution (* aop.Order.*(..))")
    public void orderMethods() { }

    @AfterThrowing(value="orderMethods()", throwing="ex")
    public void afterMethod(JoinPoint method, Throwable ex) {
       System.out.printf("Method %s threw exception : %s \n",
          method.getSignature(),ex.getClass().getName());
    }

    @Before("execution(* aop.Order.*(String))")
    public void beforePrint(String message ) {
       System.out.println("Message : " + message);
    }
```

Aspect Example

```
    @Around(value="orderMethods()")
    public void aroundMethod(ProceedingJoinPoint method)
         throws Throwable {
      System.out.println("Calling :" + method.getSignature());
      method.proceed();
      System.out.println("Completed :" +
                         method.getSignature());
    }
} // end of class
```

Ordering Aspects

- Use Order annotation to specify the order in which multiple aspects are to be applied.

```
@Component
@Aspect
@Order(0)
public class SecurityAspect {
}
```

```
@Component
@Aspect
@Order(1)
public class LogAspect {
}
```